AN1252

MIP Guidelines and Design Issues

INTRODUCTION

The purpose of this application note is to discuss information about Echelon's Microprocessor Interface Program (MIP) not available in other application notes. It is not the intention of this document to explain what the MIP is, but rather to remove the mystery from considerations of its potential uses and to offer advice regarding its implementation. Users are sometimes confused into thinking that the MIP must be used when tying a host (another processor) to the NEURON CHIP. In many and possibly most cases, the parallel I/O model will suffice in place of the MIP.

This document will contrast the MIP with an application level MIP, then provide details for helping a designer contemplating using the MIP. An application level MIP uses the parallel I/O model built into the firmware of the NEURON CHIP, and the designer must in essence write his or her own protocol to pass information to/from the host. The MIP/P50, MIP/P20, and parallel I/O model use the same hardware interface. The MIP/DPS requires a dual ported RAM.

This application note will provide a series of steps and checks necessary for the MIP to work on an MC68HC11 microprocessor. This information will prove useful for designers using other hosts as well.

Hardware interface between an MC68HC11 and NEURON CHIP will be shown, as well as example code using the parallel I/O model.

REFERENCES

Table 1. MIP Reference Documentation

Source	Title
Motorola	Parallel I/O Interface to the NEURON CHIP (AN1208)
Motorola	MC143150 / MC143120 NEURON CHIP Distributed Communications and Control Processors
Echelon	LONBUILDER Microprocessor Interface Program (MIP) User's Guide
Echelon	LONWORKS Host Application Programmer's Guide
Echelon	LONWORKS Network Interface Developer's Guide
Echelon	Serial LONTALK Adapter (SLTA/2) User's Guide

It is highly recommended that you review the application note entitled *Parallel I/O Interface to the NEURON CHIP* (AN1208). This application note describes the parallel I/O object of the NEURON CHIP, including specifics on the handshaking and token passing process used to establish synchronization and prevent bus contention. This will be a good starting point before undertaking the MIP.

Table 2. Online Services

On–Line Services	Internet Address
Motorola's Design–NET	http://motserv.indirect.com
Echelon's LonLink 415–856–7538	telnet lonlink.echelon.com (address: 198.93.128.100) ftp lonworks.echelon.com (address: 198.93.128.1) world wide page: http://www.lonworks.echelon.com

Use telnet to participate in discussions and ftp to download files and engineering bulletins. Engineering bulletins can be downloaded from the bulletin board. These bulletins are viewed using Common Ground viewer software, which can also be downloaded. The World Wide Web Home page will support both telnet and ftp connections.

The MC68HC11 files are located on Design–Net in the `miphc11.zip` file. The MC68HC3xx files are located on Design–Net and LonLink in the `mip3xx.zip` file. `mip3xx.zip` files include support for the MC68HC332, the MC68HC340, and the MC68HC360. In this document, MC68HC3xx is in reference to the MC68HC332, MC68HC340, and the MC68HC360 processors.

Echelon offers a two day class entitled *MIP and SLTA Advanced Training.*

AVAILABLE MIP PRODUCTS

As shown in Table 3, five types of MIPs are available from Echelon. Refer to the 1995 Echelon LONWORKS Products Databook for details on these products. The MIP/P20, P50, and DPS are software products. All three of these MIP products are licensed on a royalty basis from Echelon. There are no royalty fees on the first 100 copies. The MIP/P20 and P50 are packaged together. The MIP/DPS is packaged separately.

The LTS–10 and Serial LONTALK Adapter (SLTA) are sold in a single in–line module (SIM) package and external box, respectively. The LTS–10 SLTA core module is housed in a compact SIM and is used to build an SLTA. An SLTA is typically connected to a personal computer (PC). The LTS–10 and SLTA communicate to the host through a EIA–232 interface. This document references the three software MIP products, specifically the MIP/P20 and P50.

Table 3. MIP Products

MIP Products	Description
MIP/P50	MIP firmware for the 3150
MIP/P20	MIP firmware for the 3120
MIP/DPS	MIP firmware for the 3150 using Dual Port with Semaphores (i.e. Dual Ported RAM)
LTS–10	SLTA on a SIM
SLTA	Typically connected to a PC

Most of the PC interface boards made today use the MIP/P50. These include boards from the following companies:

- Echelon
- Gesytec
- Metra
- Ziatech

Echelon's SLTA contains a special MIP which communicates serially to a PC. The LONBUILDER Developer's Workbench interface board also contains a special version of the MIP firmware. A processor (such as an MC68HC11 or an MC68HC332) can be tied to the NEURON CHIP running the MIP firmware, instead of a PC.

The PC or processor connected to the NEURON CHIP is called the host processor. The MIP/P20 and MIP/P50 passes information to the host using the eleven I/O lines. The MIP/DPS uses the address lines, and the SLTA uses a Universal Asynchronous Receiver Transmitter (UART).

The MIP transfers parts of OSI layers 5 through 7 to the host. These layers mainly handle network variables and some network management of the NEURON CHIP. When using the MIP, the NEURON CHIP can be placed in either host selection or network interface selection. Typically, host selection is used. Host selection transfers the OSI layers as mentioned above to the host, increasing the number of network variables supported from 62 to 4096. NOTE: The SLTA uses host selection. In addition, all network interfaces used with the Application Programming Interface (API) must use host selection.

It is possible to run the MIP and have the NEURON CHIP do the addressing but most of the benefits of the MIP are lost, such as increasing the number of network variables. Running MIP turns the NEURON CHIP into a communication processor.

The MIP is a function call invoked in the reset "when" clause which never returns. Therefore, no other application can be used after the MIP function is called.

ADVANTAGES AND DISADVANTAGES OF USING THE MIP

Table 4 shows some of the advantages of the MIP and the application level MIP. Table 5 shows the disadvantages.

Table 4. Advantages of the MIP and Application Level MIP

MIP	Application Level MIP
Higher performance: 1–5:1 in throughput (packets/second)	
More network variables (4096 versus 62)	
	No royalty fees
	May run other applications. Not dedicated to MIP application.
	Easier to implement
Code changes are done on the host, not the NEURON CHIP	

Following are four reasons to use the MIP:

1. To increase the number of network variables from 62 to 4096.
2. To increase throughput up to five times.
3. To decrease maintenance. It will eliminate the necessity of burning an (EP)ROM or flash for the NEURON CHIP every time the application changes.
4. To use resources on the host.

One of the biggest advantages of using the MIP may be lower maintenance costs. Application code on the NEURON CHIP is typically not updated. Therefore, most of the code changes are done on the host. This reduces maintenance costs significantly by having to change code only on the host and not on the NEURON CHIP. If the code is never going to be changed, this may not be an advantage.

Echelon's MIP is provided in object code format. The application MIP may be a better choice for a simple gateway, as for example, into a foreign protocol.

Table 5. Disadvantages of the MIP and Application Level MIP

MIP	Application Level MIP
Costs in MIP and royalty fees	
No other applications can run	
	If using the MC143120, difficult to fit in other applications
	Maintenance costs in upgrading the NEURON CHIP with new code
	Uses more memory (typically RAM) to buffer up data
More difficult to implement	

The two main disadvantages of using the MIP are the cost, and the difficulty in implementation. The MIP host application is C language (C) intensive and complex. If host selection is turned on with the MIP, OSI layers 5 – 7 are transferred to the host and must be handled by the host. The long learning curve may increase development times.

Drivers are available for the PC, MC68HC3xx, and MC68HC11. The latter two come from the PC driver. All the DOS dependent code was taken out and then ported to these processors. The MC68HC3xx is documented through a `readme` file available with the MIP driver on Motorola's Design–Net or Echelon's LonLink. The MC68HC11 started with the MC68HC3xx files and then the lower level routines were changed. When developing MIP code for the MC68HC11, make use this application note, the MC68HC3xx documentation, and the MC68HC11 files on Design–Net. It should be noted that all of these files *miphcu.zip, mip3xx.zip* have NOT been fully tested and the routines to handle error conditions are left up to the user.

The DOS version has several more features than the current microprocessor versions such as being able to handle several error conditions by timing out. If needed, this will have to be added for the microprocessor's version. The I/O and various resources inside the host are set up for the specific application. It is not assumed that the user will use the application program and driver without modification. The application program and the driver are only the starting points. Except for handling time outs, the driver is set up so that it can be used with little modification.

The MIP driver code may require several thousand bytes to implement on the host, and the application code to use the driver may require even more. With all this in mind, the benefits as listed above must now be taken into consideration. Many customers have found that after proper implementation of the MIP, the time taken to learn the MIP is time well spent, and that the code is easily modified.

BUFFER USAGE

Echelon has optimized the buffer transfer from the NEURON CHIP to the output buffer by eliminating the need to write to user RAM before going to the host. Figure 1 shows the buffers in a NEURON CHIP, and Table 6 shows the buffer sequence from network to host for both the MIP and application level MIP. Host to network is the reverse step.

The sequence reading a packet from the network is: the MAC processor reads in and checks for the CRC; if the CRC is correct, the MAC processor passes the information to the network processor which checks for the address. Next, for MIP/DPS, data is sent through the external data lines to a dual ported RAM. For the MIP/P50 and MIP/P20, data is sent to the application processor then to the host. For an application level MIP, data (network variables and explicit messages) goes into memory (typically RAM) then is passed to the host. This means that an application level MIP has one more step to write than the MIP/P20 and MIP/P50, and two more steps than the MIP/DPS.

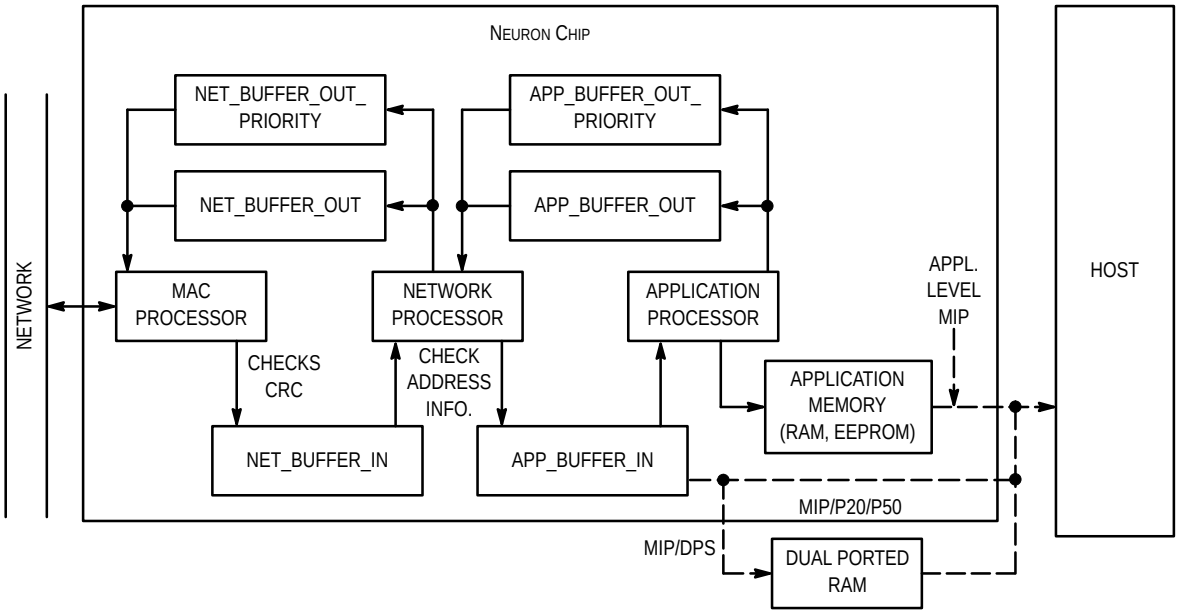

Figure 1. NEURON CHIP Buffers

Table 6. MIP versus Application Level MIP Buffer Usage

Step	MIP/P20	MIP/P50	MIP/DPS	Application MIP
1	network buffer	network buffer	network buffer	network buffer
2	application buffer	application buffer	dual port RAM / host	application buffer
3	host	host		user memory
4				host

It should be noted that an MC143120 NEURON CHIP with 1K of RAM may not have enough RAM to guarantee all of the packets on the network received. The problem is not with the MIP, but with available buffers needed by the three processors built into the NEURON CHIP. An application level MIP uses more memory to buffer the data before sending to the host.

The MC143120E2DW contains 2K of RAM and will be available in the latter half of 1995. The MC143150 has 2K of RAM on–board. With heavy traffic, 2K of RAM may still not be enough. The MC143150 has an external address/data bus which can be used to interface more RAM.

BENCHMARKS

Figure 2 shows the MIP versus Application Level MIP Performances using unacknowledged service. Table 7 shows the MIP Performance Benchmarks using unacknowledged and acknowledged services. Application overhead should bring all these numbers down. These numbers should be used only for comparison among themselves. Specific applications will depend on many factors: speed of host, network traffic, number of buffers allocated in the NEURON CHIP, and the host, to name a few.

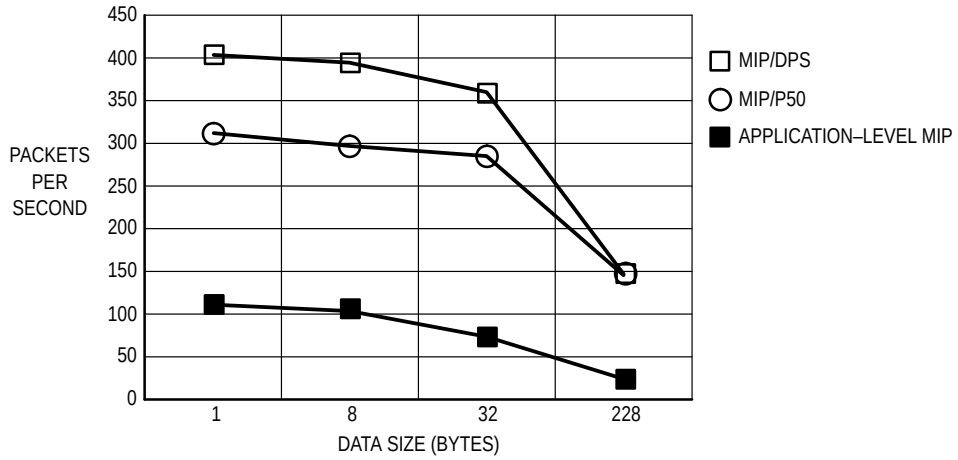

Figure 2. MIP versus Application Level, MIP Performance

Table 7. MIP Performance Benchmarks
(Using PC/386 Host at 25 MHz, Protocol Overhead of 9 bytes)

		MIP/P20	MIP/P50	MIP/DPS	SLTA and LTS–10
Unackd	1–Byte Data	205	303	404	71
	8–Byte Data	205	289	396	71
	32–Byte Data	170	260	364	56
	228–Byte Data	103	158	149	22
Ackd	1–Byte Data	76	106	106	77
	8–Byte Data	74	103	103	71
	32–Byte Data	68	94	94	59
	228–Byte Data	47	55	55	22

SUGGESTIONS FOR DEVELOPING AN MIP SYSTEM

Following are suggested steps in developing a MIP system. They are not necessarily in the order of performance, especially if a prototype is being developed to test out the feasibility of the system.

1. Determine whether the MIP is needed. A simple node may not need the MIP, whereas a node doing complex network management may benefit from it. Decide up front the requirements of the node. A prototype may be in order (with and without the MIP). Refer to the section on Advantages and Disadvantages of Using the MIP.
2. Design the overall architecture of the system. Decide which processor and software will be used. This choice will depend on factors such as speed, memory requirements, I/O, and code, just to name a few. An MC68HC3xx may be in order.
3. Pick the development tools to be used. More time was spent working around compiler, source level debugger, and target board problems than in debugging and writing code. If code is to be written code for anything but a simple MIP node, a good source level debugger is recommended. This will significantly decrease debugging time over the possibility of having to step through in assembly.

   A recommended sequence is to get your tools and your software structure (vectors, interrupts, and main program) running and debugged as quickly as possible. Before any serious debugging is done on your code, you need to be able to depend on your development tools. Remember, every software tool (compiler, linker, source level debugger, …) differs from others. Do not assume your code will work flawlessly porting from one tool/host to another.
4. Design and implement the network interface.
5. Test the hardware.
6. Design the software.
7. Test the software.

From here, as during the previous stages, good standard practices are recommended, such as software and hardware reviews.

DEVELOPMENT TOOLS

It is recommended that the tools be in place before MIP development starts. The most time consuming part of working with the MIP is not with the program, but in setting up the hardware and software to support the microprocessor used. It is therefore recommended that your tools be fully set up before any serious MIP applications development gets under way. If possible, understand your hardware and software tools before investing in them.

When porting “C” code from one compiler and/or host to another, expect to make some compiler/host dependent changes. These include:

- "C" portability: Most compilers today are ANSI C compatible. But C is not fully defined. For instance, bit fields and ordering of bits. The placement of bit 0 in a byte is compiler dependent, not "C" dependent. Typically, Motorola processor compilers place bits in a byte in the opposite order from Intel processors:

  Motorola bit 7 .. bit 0
  Intel bit 0 .. bit 7

  Echelon's available DOS MIP driver is written for a DOS machine, and care must be taken when porting to a Motorola processor. Some compilers have an option to arrange the bit order in either direction.

- Source Level Debugger (SLD): Typically the compiler and/or source level debugger are manufactured by a different company than the host emulator. Make sure the SLD supports the intended host emulator. Some SLDs use some of the resources of the host (I/O lines, software interrupts, …).

When developing code for the MIP, software and hardware support for the tools is highly recommended.

HARDWARE

Address Decode

Figure 3 shows the block diagram of the interface circuitry between the MC68HC11 and the NEURON CHIP. Figure 4 shows the detailed schematic. The address decode block addresses the NEURON CHIP as two memory registers: one for the handshaking bit to see if the NEURON is busy, and the other to pass/receive data. The NEURON CHIP is decoded at 0x8000 – 0x87ff but only addresses 0x8000 and 0x8001 are used. The MC68HC11 has a multiplexed address/data bus, and the 74HC75 is used to latch A0.

MC68HC11 to NEURON CHIP Interface Reset Circuitry

Reset signals to and from the NEURON CHIP are handled by additional logic as shown in Figure 4. There are two sources of reset for the MC68HC11 and the NEURON CHIP. One source is internally generated by the MC68HC11 or NEURON CHIP and the second source is externally generated by a Low Voltage Inhibit (LVI); for example, an MC33164 or a push–button reset switch. The MC68HC11 may reset the NEURON CHIP but not vice–versa.

Additionally, resets may come from the NEURON CHIP by means of a network management command being received over the LONWORKS network. This network management command causes the reset pin on the NEURON CHIP to become an output, and be pulsed low for a short period of time. Due to the short duration of this pulse, this reset condition must be latched (for instance, a 74HC74 D flip–flop). The output of the D flip–flop is then used to interrupt the MC68HC11 to notify the application program of this network management command. Since this signal is an interrupt to the MC68HC11, the IRQ pin must be held low until the interrupt is acknowledged by the interrupt service routine. The interrupt is then cleared by setting PD2 I/O pin low and restoring it back high in the interrupt service routine. Optionally, in case of multiple IRQ interrupts, the output of the flip–flop may also be used as an input to another I/O pin (such as PD4) so that the interrupt service routine may determine the source of the IRQ interrupt.

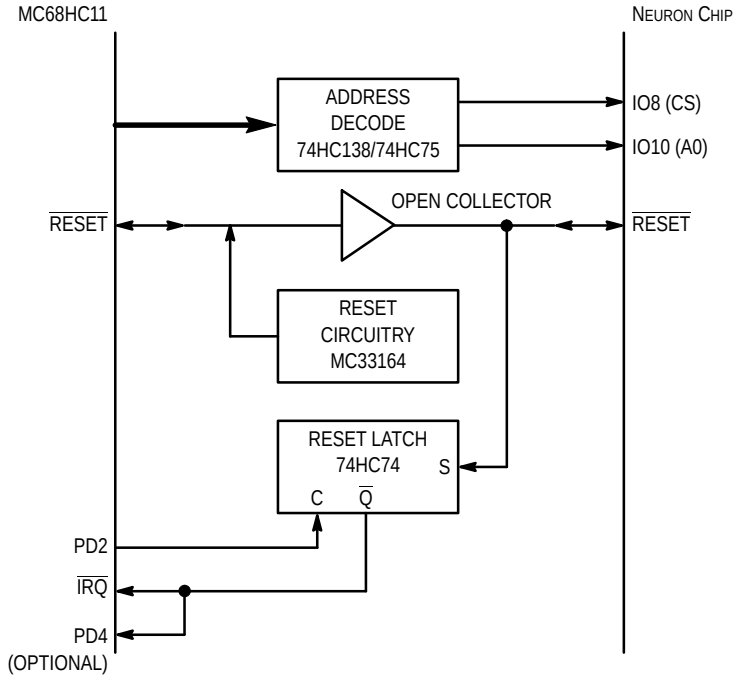

Figure 3. MC68HC11 to NEURON CHIP Interface Block Diagram

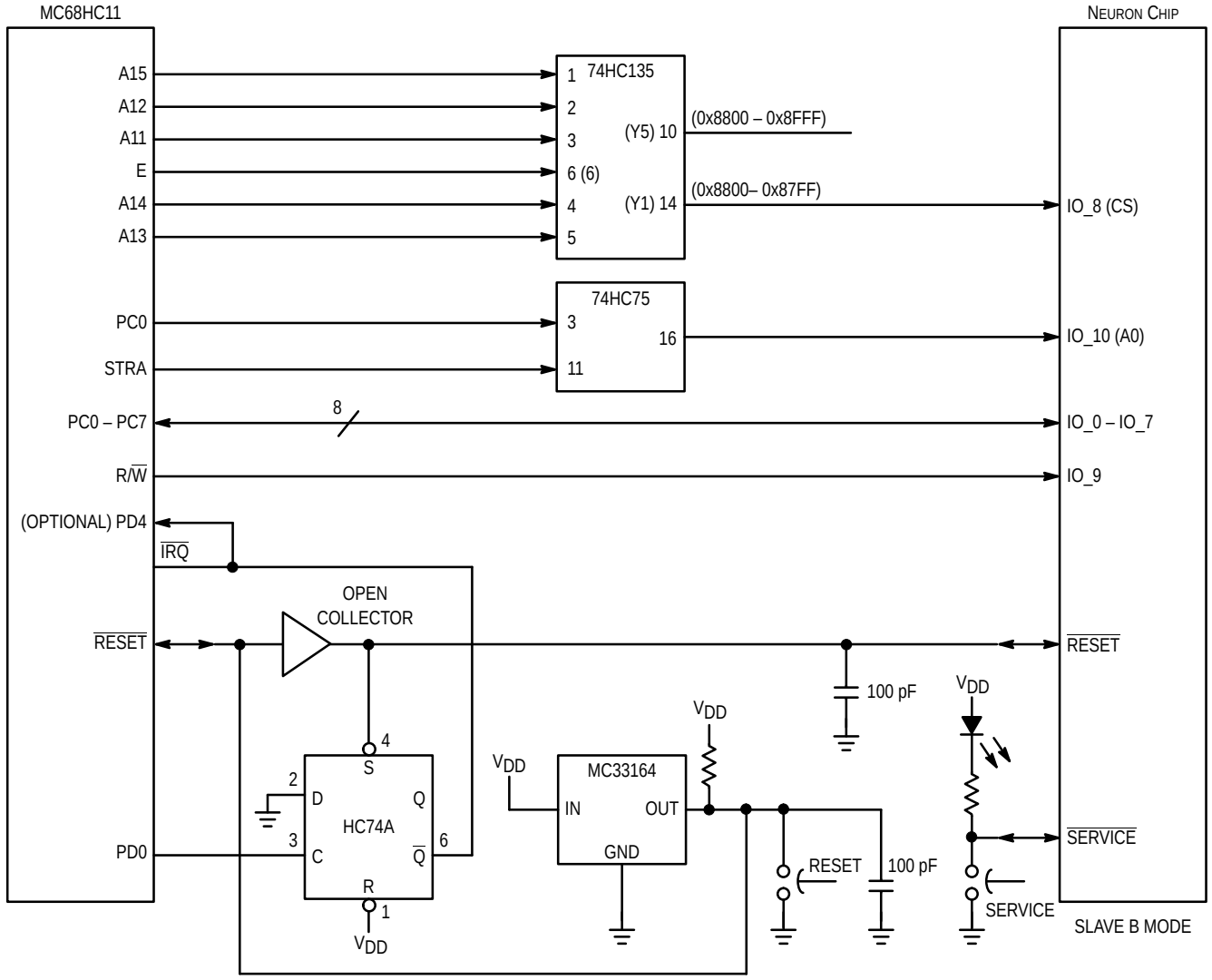

Figure 4. MC68HC11 to NEURON CHIP Interface

The open collector device between the MC68HC11 reset pin and the NEURON CHIP reset pin is used to prevent a NEURON CHIP source reset from resetting the MC68HC11. When designing the reset circuit the following factors must be taken into consideration:

- How much current the NEURON CHIP can source.
- The saturation voltage of the LVI. This voltage will be current dependent.
- The voltage level the NEURON CHIP will reset.
- The voltage level the NEURON CHIP will output at reset.
- The current level at which any LEDs will turn on.
- Voltage drops across all components, including diodes and resistors.
- Any time constants (ex: RC networks).
- Saturation voltage of the open collector device.

SOFTWARE

Debug and Initial Set–up

It is recommended that the hardware interface be debugged and the tools set up before any software effort is started. It is useful to attach a logic analyzer to the NEURON CHIP's eleven I/O pins in order to understand the relationship between the NEURON CHIP and the host. If an MIP is going to be used, first debug the hardware interface using only the parallel I/O model.

MC68HC11 and NEURON CHIP Using an Application Level MIP

Exhibits 1 and 2 show a sample NEURON C and MC68HC11 program, respectively, using a NEURON CHIP's parallel I/O model. No MIP is used. The programs continuously pass data back and forth. The MC68HC11 sends eight bytes of data (0x50 – 0x57) to the NEURON CHIP, which then sends four bytes of data back (0x0 – 0x3). This is continuously repeated.

Exhibit 3 shows the results of using a logic analyzer on the NEURON CHIP's eleven I/O lines, triggering off of the rising

edge of CS. For the MC68HC11, Exhibits 1, 2, and 3 can be used as a guide to expectations for software, hardware, and debugging of an application level MIP.

MIP/P20, P50 Driver

The DOS MIP driver was originally created by Echelon to run on a DOS machine using Echelon's Serial LONTALK Adapter (SLTA). It was modified for the MC68HC11 microprocessor. Echelon wrote the host application and driver programs to demonstrate the use of a host microprocessor, in their case a PC, using an SLTA or MIP. The SLTA uses a special version of the MIP firmware. Instead of using a parallel interface from the NEURON CHIP, it uses a UART to provide serial data out. Echelon documents their host application program in a manual sent with the SLTA or MIP products.

The MIP application is a function call invoked in the *reset when* clause and when called, never returns. Therefore, no other application can be used after the MIP function is invoked.

The MIP driver on the MC68HC11 is written mostly in "C", the rest in assembly for handling interrupts, start–up files, and some of the I/O functions. There is a "tick" timer, typically in the range of 30 ms to 200 ms, which allows for the MC68HC11 application to read and write buffers to the MC68HC11 MIP driver.

MIP DETAILS

Host software is divided into two parts: the driver and the host application. The driver handles buffering packets and the interface to the NEURON CHIP. This driver also ensures that the sequence of calls to the MIP and function calls from the host application are correct.

The driver will:

- Handle buffer request/response mechanism so application will not have to track it.
- Handle difference between application layer and link layer protocols. Our drivers will support only one application program.

The interface between the host application and the driver is called the Application–Layer Interface. The Application–Layer Interface passes parameters back and forth between the host application and the driver. For overall usage of the MIP driver, refer to the *LONBUILDER Microprocessor Application Programmer's Guide*. There are four function calls between the host application and the driver:

open: initialize parameters. This call is used to allocate resources for operation of the driver and prepare the MIP interface to transfer data. This is typically called at the start of the program.

close: opposite of open. Deallocates resources used by the MIP driver. Typically not called or called if the program ends.

read: application reads data passed from the NEURON CHIP from the driver's buffers.

write: application writes data to be sent to the NEURON CHIP to the driver's buffers.

Formats of the data field being passed to or from the NEURON CHIP are outlined in the *LONWORKS Host Application Programmer's Guide*. Additional information on network management commands and addressing structure formats are in Motorola's DL159 *LONWORKS Technology Device Data*, Appendix A.

The interface between the driver and the MIP is called Link–Layer Interface. Data is sent and received with the sequence of events between the driver and the MIP. There are two types of commands sent to the MIP, commands that stay local to the NEURON CHIP (*niComm* command) and those that go out over the network (*niNetmgmt* command). Also a queue priority from the NEURON CHIP must be requested (*TQ:* transaction queue with a response to it, *NTQ:* non transaction queue which uses unacknowledged service, *TQP:* with priority, *NTQP:* without priority).

MODIFYING THE MIP

Network management commands not handled by the NEURON CHIP must be handled by the host. The user must save data passed in some of these commands and also respond to the Network Manager with the information requested by other network management commands.

In order to send a message over the MIP interface, the data must be enclosed with appropriate MIP header information. The header information includes length of the data and the addressing information for the message destination. The application software to handle these network management commands must be written.

To make this operation easier, one may use an example provided by Echelon to handle this operation. This example is included with the MIP product and also is available on the LonLink bulletin board service. The example consists of a series of "C" language source code files starting with the `HA.C` file. `HA.C` implements a very specific example for updating and displaying network variables, sending and receiving messages, and binding to other nodes through a DOS based console. With some modification, some of the other functions called by `HA.C` may be used.

Files `NI_MSG.C` and `APPLMSG.C` may be modified, compiled, and linked with user code. The file `NI_MSG.C` contains user callable functions of *ni_init*, `ni_send_msg_wait`, `ni_receive_msg`, and `ni_send_response`. As their name implies, these functions may be used to initialize the network interface, send a properly formatted message over the interface, receive a message over the interface, and send a response over the MIP interface.

The file `APPLMSG.C` contains code to handle the network management commands to which the host computer must respond. These commands are `query_nv_config`, `nv_fetch`, `update_nv_config`, `query_snvt`, and `set_node_mode`.

Used with the above mentioned files are two c header files. These are `NI_MSG.H` and `NI_MGMT.H`. These files contain structure definitions and must be included in your C source file. NOTE: The data type definitions of bits is little endian bit ordering and must be reversed to big endian bit ordering for Motorola microcontroller designs.

EXHIBIT 1
NEURON CHIP CODE USING PARALLEL I/O MODEL

```
/*********************************************************************
 Example program for a NEURON CHIP in parallel I/O interface with a
 MC68HC11. The Neuron chip is in slave B mode and the HC11 is acting
 as a master. The program enters in an infinite loop of read and write
 cycles.
*********************************************************************/

#define maxin 10
IO_0 parallel slave_b p_bus;

unsigned char i=0;                    // counter to fill buffer
unsigned int len_out=4;     //number of bytes for input and output

struct parallel_io
{
   unsigned char len;                 // actual number of bytes in buffer
   unsigned char buf[maxin];          // array to store data
}pio;                                 // name of structure

when (io_in_ready(p_bus))
{
   pio.len = maxin;                   // maximum input length
   io_in(p_bus,&pio);                 // read in data
   io_out_request(p_bus);             // request to output
}

when (io_out_ready(p_bus))
{
   pio.len=len_out;                   // number of bytes to be output
   for (i=0; i<len_out; i++)          // fill buffer with data
     pio.buf[i] = i;
   io_out(p_bus,&pio);                // output data
}
```

EXHIBIT 2
MC68HC11 CODE TO INTERFACE TO NEURON CHIP USING PARALLEL I/O MODEL

```
/*     description: Use yes2.nc on a LB emulator.
                Transmits: 00, 01
                           length = 8
                           data = $50 – 57
                Receives:  00, 01
                           length = 4
                           data = 0,1,2,3
 */

/*****************************************************************
    Example program for a MC68HC11 interfacing with a Neuron chip.  The
    NEURON CHIP is in parallel I/O slave B mode and the HC11 is acting
    as a master.  The program synchronizes the HC11 master and Neuron
    chip slave and then enters an infinite loop of read and write
    cycles.
*****************************************************************/

#define HS_MASK 0x01               /* mask for lSBit of control register*/
#define CMD_RESYNC 0x5A            /* initial command to synchronize neuron
                                      chip */
#define CMD_ACKSYNC 0x07           /* synchronization acknowledge from
                                      slave */
#define CMD_XFER 0x01              /* command to transfer data */
#define LENGTH_OUT 0x08            /* length of data transfer from master*/
#define EOM 0x00                   /* end of message */
#define MAX_ 0x09                  /* maximum size of data buffer */
#define DATA_REGISTER 0x8000       /* even address accesses data register*/
#define CONTROL_REGISTER 0x8001    /* odd address accesses handshake
                                      register*/
#define MASTER 1                   /* token tracking for master write */
#define SLAVE 0                    /* token tracking for master read */
unsigned char token;               /* tracks read and write cycles */
unsigned char *datareg, *hs;       /* pointers for data and handshake
                                      registers */
struct parallel_io                 /* buffer for data transfers*/
{
 unsigned char len;                /* length of data transferred */
 unsigned char data[MAX_];         /* array to store data */
}pio;

/*****************************************************************
   Verify the processors are synchronized before any data is
   transmitted.  The master sends the command to resynchronize until the
   slave acknowledges with CMD_ACKSYNC.  The master owns the token after
   resynchronization.
*****************************************************************/
```

```
sync_loop()
{
   while (*datareg != CMD_ACKSYNC) {        /* loop until acknowledge
                                               received */
      hndshk();
      *datareg = CMD_RESYNC;                /* send command to resync */
      hndshk();
      *datareg = EOM;                       /* send end of message */
      hndshk();
   }
   token = MASTER;                 /* master owns token after reset */
}

/*********************************************************************
    Verify the slave is ready for the next byte transaction.  Read the
    control register of the slave which accesses the handshake signal
    (least significant bit of the control register).  Mask all bits but
    the handshake bit and verify if the handshake signal has gone low.
*********************************************************************/

hndshk()       /* infinite loop until the handshake bit goes low */
{
   while ((*hs) & HS_MASK);
}

/*********************************************************************
    Identify the owner of the token to determine if a read or write is
    appropriate. If the master owns the token a write cycle is
    performed;  if the slave owns the token a read cycle is initiated.
    This process prevents bus contention, as only the owner of the token
    can write to the bus.
*********************************************************************/

main_loop()
{
   while(1) {                      /* infinite loop of read/write cycles */
      if (token == MASTER)         /* master owns the token */
         write();                  /* master writes to the slave */
      else                         /* slave owns the token */
         read();                   /* master reads from the slave */
   }
}
```

```
/*********************************************************************
   The master owns the token at the start of this function, therefore,
   the master can write to the bus.  The buffer is filled, the command
   to send data (CMD_XFER) is transmitted, the length (number of bytes
   of data) is transmitted and the data is transmitted one byte at a
   time.  The handshake signal is monitored for low transition before
   each byte transfer.  After the data is transmitted, the token is
   processed.
*********************************************************************/

write()
{
    unsigned char send_data;
    make_buffer();                  /* assign length and create data */
    hndshk();
    *datareg = CMD_XFER;            /* command to send data */
    hndshk();
    *datareg = pio.len;             /* send length of data to be
                                       transmitted */
    for (send_data=0; send_data<pio.len; send_data++) {
       hndshk();
       *datareg = pio.data[send_data];       /* send data one byte at a time */
    }
    pass_token();                   /* process the token */
}

/*********************************************************************
    Assign the data length.  Fill the buffer with data before
    transmitting.  The data is ascii: P,Q,R,S,T,U,V,W.
*********************************************************************/

make_buffer()
{
    unsigned char data_out;           /* counter for creating data */
    pio.len = LENGTH_OUT;             /* length of bytes of data */
    for(data_out=0; data_out<LENGTH_OUT; data_out++)
    pio.data[data_out]=(data_out+(0x50));         /* ascii output */
}

/*********************************************************************
    The slave has the token at the beginning of this function,
    therefore, the master reads from the slave.  If the first byte is
    the command to transfer, read the length of data bytes to be
    received, read each byte of data, then transfer the token the
    master.  If the slave has no data to send, assume the command is a
    NULL and simply transfer the token to the master.  Always wait for
    the handshake signal to be low before each transaction.
    Note: No error checking is implemented to verify the command is a
    NULL.
*********************************************************************/
```

```
read()
{
   unsigned char cmd;              /* stores the command from the slave */
   unsigned char i=0;              /* counter to read in data */
   hndshk();
   if ((cmd = *datareg) == CMD_XFER) {      /* slave has data to send */
      hndshk();
      pio.len = *datareg;              /* read length of data to be
                                          transferred */
      while (pio.len--) {          /* read in each byte of data */
         hndshk();
         pio.data[i]=*datareg;         /* put data in a buffer */
         ++i;
      }
   }
   pass_token();                   /* pass token to the master */
}

/*********************************************************************
    Process the token.  If the master owns the token, send an end of
    message to the bus and then pass the token to the slave. If the
    slave owns the token, simply pass the token to the master.
*********************************************************************/

pass_token()
{
   if (token == MASTER) {              /* master owns the token */
      hndshk();
      *datareg = EOM;                  /* write an end of message */
      token = SLAVE;                   /* pass the token to the slave */
   }
   else                                /* slave owns the token */
      token = MASTER;                  /* pass the token to the master */
}

main()
{
   datareg = (unsigned char*) DATA_REGISTER;       /* data pnts to the data
                                                      reg */
   hs = (unsigned char*) CONTROL_REGISTER;         /* hs pnts to the cntrl
                                                      reg */
   sync_loop();                        /* synchronize the processors */
   main_loop();                        /* infinite loop of read/write
                                          cycles */
}
```

EXHIBIT 3
LOGIC ANALYZER READINGS FOR MC68HC11/NEURON CHIP USING PARALLEL I/O MODEL

Loc.	D7 – D0	CS	R/W	A0	Description
–4	01	0	1	1	
–3	01	0	1	1	D = 1, never busy
–2	00	0	1	1	D = 0, Neuron ready
–1	5A	0	0	0	Write Cmd resync (5A)
Trig	5A	0	0	0	
1	1	0	1	1	D = 1, NC busy
2	1	0	1	1	
3	:	:	:	:	repeated
:	1	0	1	1	
16	00	0	1	1	D + 0, NC ready
17	00	0	1	0	
18	00	0	1	0	HC11 write EOM (00)
19	01	0	0	1	D = 1, NC busy
20	:	:	:	:	repeated
:	06	0	1	1	D = 0, NC ready
29	07	0	1	0	NC read cmd Acksync
30	01	0	1	0	Note: This should have been 0 (EOM)
31	00	0	1	1	D = 0, NC ready
32	01	0	0	0	HC11 write Cmd_Xfer (01)
33	01	0	1	1	
34	:	:	:	:	repeated
:	01	0	1	1	
131	08	0	0	0	HC11 write length (8)
132	01	0	1	1	D = 1, NC busy
133	:	:	:	:	repeated
:	01	0	1	1	
159	00	0	1	1	D = 0, NC ready
160	50	0	0	0	HC11 write data (50)
161	00	0	1	1	D = 0, NC ready
162	51	0	0	0	(51)
163	00	0	1	1	
164	52	0	0	0	(52)
165	00	0	1	1	
166	53	0	0	0	(53)
167	00	0	1	1	
168	54	0	0	0	(54)
169	01	0	1	1	
170	55	0	0	0	(55)
171	00	0	1	1	
172	56	0	0	0	(56)
173	00	0	1	1	
174	57	0	0	0	(57)

Loc.	D7 – D0	CS	R/W	A0	Description
175	00	0	1	1	
176	00	0	1	0	
177	00	0	0	0	HC11 write EOM (O)
178	01	0	1	1	D = 1, NC busy
179	:	:	:	:	repeated
:	01	0	1	1	
397	01	0	1	0	HC11 Reads CMD–XFER (01)
398	04	0	1	1	D = 0, NC ready
399	04	0	1	1	
400	04	0	1	0	
401	04	0	1	1	HC11 Reads length (4)
402	00	0	1	0	D = 0, NC ready
403	00	0	1	1	HC11 Reads Data (00)
404	00	0	1	1	D = 0, NC ready
405	01	0	1	0	
406	02	0	1	1	HC11 Reads data (01)
407	02	0	1	1	D = 0, NC ready
408	02	0	1	0	
409	02	0	1	1	HC11 Reads data (02)
410	02	0	1	0	D = 0, NC ready
411	02	0	1	0	
412	03	0	1	1	
413	01	0	0	0	HC11 Reads data (03)
414	01	0	1	1	HC11 Write Cmd—xfer
415	01	0	1	1	D = NC busy
:	:	:	:	:	
496	01	0	1	1	D = NC busy